Defusing a Binary Bomb

**1 Introduction**

The nefarious Dr. Evil has planted a slew of “binary bombs” on our class machines. A binary bomb is a program that consists of a sequence of phases. Each phase expects you to type a particular string on stdin. If you type the correct string, then the phase is defused and the bomb proceeds to the next phase. Otherwise, the bomb explodes by printing "BOOM!!!" and then terminating. The bomb is defused when every phase has been defused. There are too many bombs for us to deal with, so we are giving each student a bomb to defuse. Your mission, which you have no choice but to accept, is to defuse your bomb before the due date. Good luck, and welcome to the bomb squad!

**Step 1: Get Your Bomb**

You can work alone on this assignment or work with someone else. If you work with someone else, take turns being in charge of the keyboard.

You can obtain your bomb by pointing your Web browser at:

[http://student2.cs.appstate.edu:15002](http://student2.cs.appstate.edu:15002/)

This will display a binary bomb request form for you to fill in. Enter your user name and email address and hit the Submit button. **If you want to work with someone else, enter the concatenation of your user names (example: norriscafenwickjb) and either email address.** The server will build your bomb and return it to your browser in a tar file called bombk.tar where k is the unique number of your bomb.

Create a lab1 subdirectory of your 3482 directory on the student2 machine. You’ll want to move your bombk.tar file to this directory. If you are working with someone else, you'll put the tar file in both of your directories so you can both play around with it.  Get into your lab1 directory and give the command:

*tar -xvf bombk.tar .*

This will create a directory called ./bombk with the following files:

* README : Identifies the bomb and its owners.
* bomb : The executable binary bomb.
* bomb.c : Source file with the bomb’s main routine and a friendly greeting from Dr. Evil.

If for some reason you request multiple bombs, this is not a problem. Choose one bomb to work on and delete the rest.

**Step 2: Defuse Your Bomb**

Your job for this lab is to defuse your bomb. You must do the assignment on the student2 machine. In fact, there is a rumor that Dr. Evil really is evil, and the bomb will always blow up if run elsewhere. There are several other tamper-proofing devices built into the bomb as well, or so we hear.

You can use many tools to help you defuse your bomb. Please look at the hints section for some tips and ideas. The best way is to use your favorite debugger, gdb, to step through the disassembled binary.

Each time your bomb explodes it notifies the bomblab server, and you lose 1/2 point (up to a max of 20 points) in the final score for the lab. So there are consequences to exploding the bomb. You must be careful!

The first four phases are worth 10 points each. Phases 5 and 6 are a little more difficult, so they are worth 15 points each. So the maximum score you can get is 70 points.

Although phases get progressively harder to defuse, the expertise you gain as you move from phase to phase should offset this difficulty. However, the last phase will challenge even the best students, so please don’t wait until the last minute to start.

The bomb ignores blank input lines. If you run your bomb with a command line argument, for example,

*./bomb psol.txt*

then it will read the input lines from psol.txt until it reaches EOF (end of file), and then switch over to stdin . In a moment of weakness, Dr. Evil added this feature so you don’t have to keep retyping the solutions to phases you have already defused.

You can run it in the debugger, providing it a file that contains the solutions by typing:

r psol.txt

from within the debugger (r = run).

To avoid accidentally detonating the bomb, you will need to learn how to single-step through the assembly code and how to set breakpoints. You will also need to learn how to inspect both the registers and the memory states. One of the nice side-effects of doing the lab is that you will get very good at using a debugger. This is a crucial skill that will pay big dividends the rest of your career.

**Handin**

There is no explicit handin. The bomb will notify your instructor automatically about your progress as you work on it. You can keep track of how you are doing by looking at the class scoreboard at:

[http://student2.cs.appstate.edu:15002/scoreboard](http://student2.cs.appstate.edu:15002/scoreboard%20)

This web page is updated continuously to show the progress for each bomb.

**Hints (Please read this!)**

There are many ways of defusing your bomb. You can examine it in great detail without ever running the program, and figure out exactly what it does. This is a useful technique, but it not always easy to do. You can also run it under a debugger, watch what it does step by step, and use this information to defuse it. This is probably the fastest way of defusing it.

We do make one request, please do not use brute force! You could write a program that will try every possible key to find the right one. But this is no good for several reasons:

* You lose 1/2 point (up to a max of 20 points) every time you guess incorrectly and the bomb explodes.
* Every time you guess wrong, a message is sent to the bomblab server. You could very quickly saturate the network with these messages, and cause the system administrators to revoke your computer access.
* We haven’t told you how long the strings are, nor have we told you what characters are in them. Even if you made the (incorrect) assumptions that they all are less than 80 characters long and only contain letters, then you will have 2680 guesses for each phase. This will take a very long time to run, and you will not get the answer before the assignment is due.

There are many tools that are designed to help you figure out both how programs work, and what is wrong when they don’t work. Here is a list of some of the tools you may find useful in analyzing your bomb, and hints on how to use them.

* gdb: The GNU debugger, this is a command line debugger tool available on virtually every platform. You can trace through a program line by line, examine memory and registers, look at both the source code and assembly code (we are not giving you the source code for most of your bomb), set breakpoints, set memory watch points, and write scripts. A handy cheat sheet can be found on the asulearn page below the lab assignment.  
    
  Here are some other tips for using gdb:
  + To keep the bomb from blowing up every time you type in a wrong input, you’ll want to learn how to set breakpoints.
  + For online documentation, type “ help ” at the gdb command prompt, or type “ man gdb ”, or “ info gdb ” at a Unix prompt. Some people also like to run gdb under gdb-mode in emacs.
* *objdump -t*  
  This will print out the bomb’s symbol table. The symbol table includes the names of all functions and global variables in the bomb, the names of all the functions the bomb calls, and their addresses. You may learn something by looking at the function names!
* *objdump -d*  
  Use this to disassemble all of the code in the bomb. You can also just look at individual functions. Reading the assembler code can tell you how the bomb works. Although objdump -d gives you a lot of information, it doesn’t tell you the whole story. Calls to system-level functions are displayed in a cryptic form. For example, a call to sscanf might appear as: 8048c36: e8 99 fc ff ff call 80488d4 <\_init+0x1a0> To determine that the call was to sscanf , you would need to disassemble within gdb .
* *strings*  
  This utility will display the printable strings in your bomb.

Looking for a particular tool? How about documentation? Don’t forget, the commands apropos, man, and info are your friends. In particular, man ascii might come in useful. info gas will give you more than you ever wanted to know about the GNU Assembler. Also, the web may also be a treasure trove of information. If you get stumped, feel free to ask your instructor for help.

**Okay, some more hints**

* Defusing a phase means you need to figure out what input to give that phase that prevents it from calling explode bomb. Take a look at the main in bomb.c and you'll see that it consists of code that calls read\_line and then passes the input read (a C-string) to a phase.  Each of you has a different bomb so the input that works for one of you may not work for someone else.
* You do not necessarily need to completely understand each line of the assembly code of a phase to figure out the input that is needed. You need to look at locations in memory and registers to find pieces of your input and how those pieces are used. Rather than deciphering the assembly code line by line, you’ll likely defuse each phase faster by looking for those lines of code that manipulate your input.
* You can use gdb in a variety of useful ways:

*convert from decimal to hex: p/x 23*

*convert from hex to decimal: p/d 0x17*

*look at a string in memory: x/s 0x804a170*

*look at a word in memory (see value in hex): x/x 8 + $ebp*

*look at 20 words in memory (see values in hex): x/20x 0x804a1e0*

*look at a value in a register (see value in hex): p/x $eax*

*break at a specific line of machine code: b \* 0x8048c27*

*break at the beginning of a function: b phase\_2*

*step to the next line of machine code: stepi*

*execute but don’t step into called function: nexti*

* In some phases, a C library function is called on your input. Use google to find out what that function does and use gdb to examine the function’s parameters. This will help you determine the type of input that is expected for that phase. Don't use gdb to step into the function. You can use "nexti" to execute but not step into.
* Although you don’t want to use brute-force, if you figured out that the value you need is within some finite range, it is fine to try each value in that range until you find the one that keeps the bomb from exploding.
* Each phase corresponds to some high-level C code construct (such as a while-loop). If you can figure out the C code construct, it will be easier for you to then decipher the assembly code.